

Code Reuse Through Hierarchies

You will learn about different ways of creating class hierarchies to better organize and group attributes and methods in order to facilitate code reuse

James Tam

Real Life Hierarchies



- Base entity:
- Attributes: age, height, weight
 - Actions: eat(), sleep(), excrete(), multiply()



- Derived entity: "martial artist" has all attributes and actions of base entity plus
- Attributes: belt/level
 - Actions: ironPalmStrike(), shadowlessKick()



- Derived entity: 'spy' has all attributes and actions of base entity plus
- Attributes: territory, code name e.g. 0-0TAM
 - Actions: stealth(), codebreaking(), lockPicking()

James Tam

Review: Association Relation Between Classes

- One type of association relationship is a ‘**has-a**’ relation (also known as “aggregation”).
 - E.g. 1, A car <has-a> engine.
 - E.g. 2, A lecture <has-a> student.
- Typically this type of relationship exists between classes when a class is an attribute of another class.

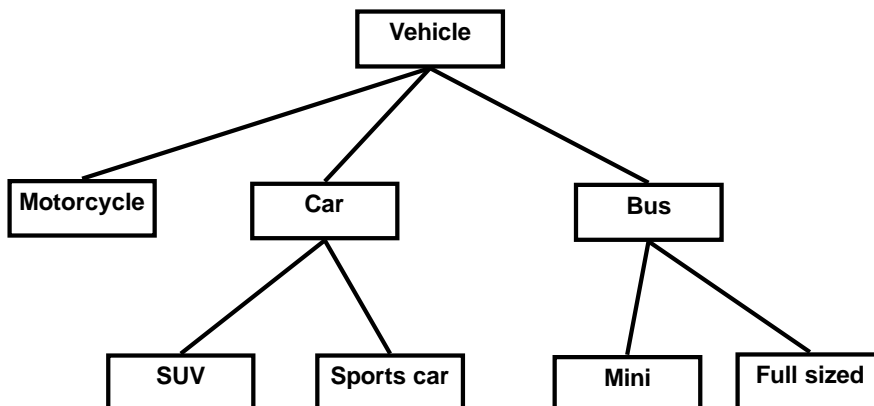
```
public class Car {  
    private Engine anEngine;  
    private Lights carLights;  
    public start() {  
        anEngine.ignite();  
        carLight.turnOn();  
    }  
}
```

```
public class Engine {  
    public boolean ignite() {  
        .. }  
}  
  
public class Lights {  
    private boolean isOn;  
    public void turnOn() {  
        isOn = true;}  
}
```

James Tam

A New Type Of Association: Is-A (Inheritance)

- An inheritance relation exists between two classes if one class is a more specific variant type of another class

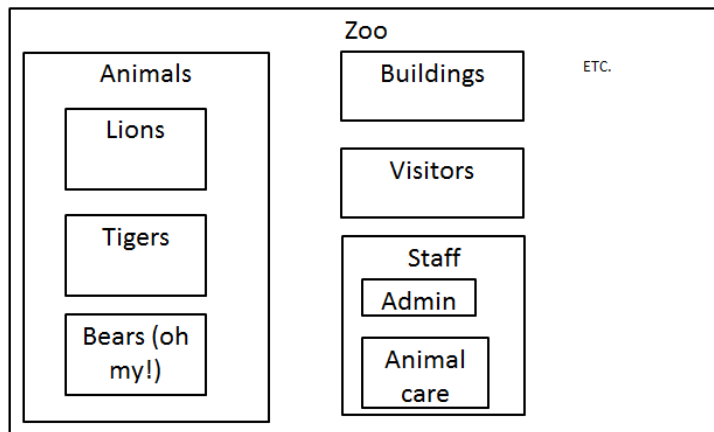


James Tam

Recall O-O Approach: Finding Candidate Classes

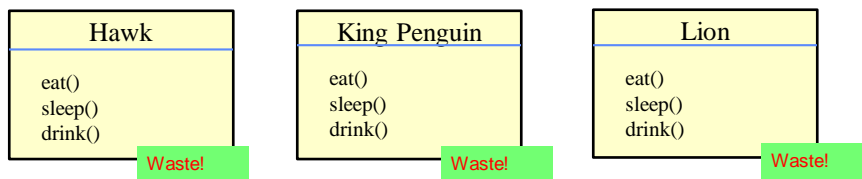
An Example Of The Object-Oriented Approach (Simulation)

- Break down the program into entities (classes/objects - described with *nouns*)



What If There Are Commonalities Between Classes

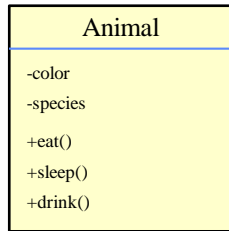
- Examples:
 - All birds 'fly'
 - Some types of animals 'swim': fish, penguins, some snakes, crocodiles, some birds
 - All animals 'eat', 'drink', 'sleep' etc.
 - Under the current approach you would have the same behaviors repeated over and over!



James Tam

New Technique: Inheritance

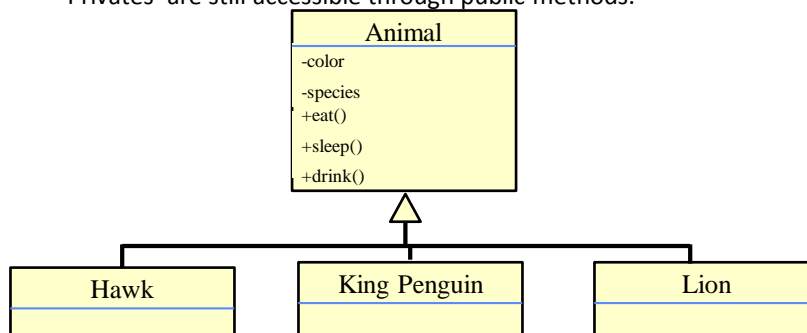
- When designing an Object-Oriented program, look for common behaviors and attributes
 - E.g., color, species, eat(), drink(), sleep()
 - These commonalities are defined in a 'parent' class



James Tam

New Technique : Inheritance (2)

- These commonalities are defined in a 'parent' class
 - Classes that are derived from (are more specific versions) of the parent class are referred to a 'child' classes.
- As appropriate other 'child' classes will directly include or 'inherit' all the non-private attributes and behaviors of the parent class.
 - 'Privates' are still accessible through public methods.



James Tam

Defining A Class That Inherits From Another

Format:

```
public class <Name of child class> extends <Name of parent class>
{
    // Definition of child class - only what is unique to
    // this class
}
```

This means that a **Lion** object **AUTOMATICALLY** has all the capabilities of an **Animal** object

Example:

```
public class Lion extends Animal
{
    public void roar() {
        System.out.println("Rawr!");
    }
}
```

The only attributes and methods that need to be specified are the ones unique to a lion

James Tam

First Inheritance Example

- Location of the full online example:
`/home/219/examples/hierarchies/1basicExample`

James Tam

Class Person



Image of James
Tam courtesy of
James Tam

```
public class Person
{
    private int age;

    public Person() {
        age = 0;
    }

    public Person(int anAge) {
        age = anAge;
    }

    public void doPersonStuff() {
        System.out.println("Eat, sleep, drink, excrete, be" +
            " fruitful");
    }
}
```

James Tam

Class Hero: A Hero Is A Person

```
public class Hero extends Person {
```

This automatically gives
instances of class Hero all
the capabilities of an instance
of class Person

```
}
```

James Tam

Class Hero: A Person But [A Whole Lot More](#)

```
public class Hero extends Person
{
    private int heroicCount;

    public Hero()
    {
        heroicCount = 0;
    }

    public void doHeroStuff()
    {
        System.out.println("Saving the world for: truth!, " +
            " justice!, and all that good " +
            " stuff!");

        heroicCount++;
    }
}
```



Image of super
James Tam courtesy
of James Tam

James Tam

The Driver Class: **Person** Vs. **Hero**

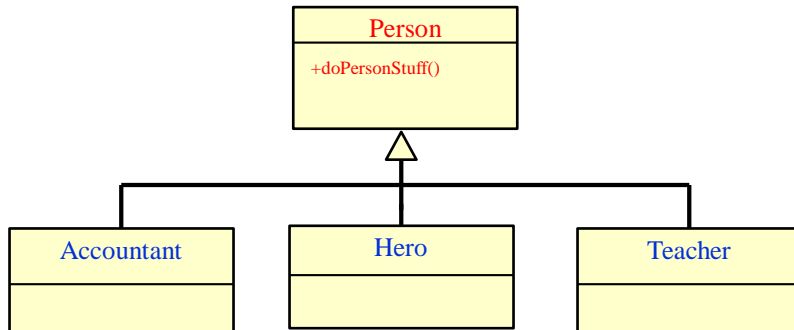
```
public class Driver
{
    public static void main(String [] args)
    {
        Person bob = new Person():
        bob.doPersonStuff(); Eat, sleep, drink, excrete, be fruitful
        System.out.println();

        Hero clark = new Hero():
        clark.doPersonStuff(); Eat, sleep, drink, excrete, be fruitful
        clark.doHeroStuff();
        Saving the world for: truth!, justice!, and all that good stuff
    }
}
```

James Tam

Benefit Of Employing Inheritance

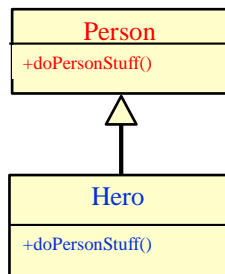
- Code reuse:
 - The common and accessible attributes and methods of the **parent** will automatically be available in all the **children**.



James Tam

New Terminology: Method Overriding

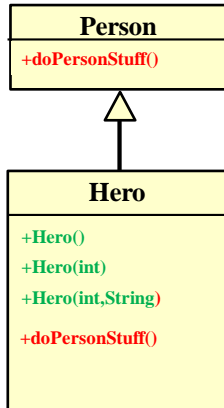
- Overriding occurs when the **parent** class has a different version of a method than was implemented in the **child** class.



James Tam

Reminder: Method Overriding Vs. Method Overloading

- **Method overriding**: different versions of a method in parent vs. child
- **Method overloading**: different versions of a method in a single class definition



James Tam

Method Overriding Example

- Location of the complete example:
- /home/219/examples/hierarchies/2overriding

James Tam

Class Hero

```
public class Hero extends Person
{
    // New method: the rest of the class is the same as the
    // previous version
    public void doPersonStuff()
    {
        System.out.println("Pffff I need not go about " +
                           "mundane nicities such as eating!");
    }
}
```

James Tam

The Driver Class (Included For Reference)

```
public class Driver
{
    public static void main(String [] args)
    {
        Person bob = new Person();
        bob.doPersonStuff(); Eat, sleep, drink, excrete, be fruitful
        System.out.println();

        Hero clark = new Hero();
        clark.doPersonStuff();
        Pffff I need not go about mundane nicities such as eating!
    }
}
```

James Tam

Overriding: Who Do We Call?

- `bob.doPersonStuff();`
- `clark.doPersonStuff();`

James Tam

New Term: Binding

- When a reference and a method are specified together, binding determines which version of the method is called.
- If neither method overloading nor method overriding are employed then binding is very easy to determine.

```
Person jim = new Person();  
  
jim.setAge(27);  
  
public class Person {  
    private int age;  
    public Person() {  
        age = 0;  
    }  
    public setAge(int anAge) {  
        age = anAge;  
    }  
}
```

James Tam

New Term: Binding (2)


- Early binding (overloading): determined at compile time (by 'javac')
 - Parameter list determines
- Late binding (overriding): determined at run time (by 'java')
 - The type of the implicit parameter ("this" reference) determines

James Tam

Method **Overloading** Vs. Method Overriding

- Method Overloading (what you should know)
 - Multiple method implementations for the same class
 - Each method has the same name but the type, number or order of the parameters is different (signatures are not the same)
 - The method that is actually called is determined at program *compile time* (**early binding**).
 - i.e., <reference name>.<method name>(parameter list);

Distinguishes
overloaded methods



James Tam

Method **Overloading** Vs. Method Overriding (2)

- Examples of method overloading:

```
public class Foo
{
    public void display( ) { }
    public void display(int i) { }
    public void display(char ch) { }
}

Foo f = new Foo ();
f.display( );
f.display(10);
f.display('c');
```

Binding at compile time (early)

James Tam

Method Overloading Vs. Method **Overriding** (3)

- Method Overriding

- The method is implemented differently between the parent and child classes.
- Each method has the same return value, name and parameter list (identical signatures).
- The method that is actually called is determined at program *run time* (late binding).
- i.e., <reference name>.<method name> (parameter list);

The type of the reference (implicit parameter "this") distinguishes overridden methods

James Tam

Example Overriding: The Type Of The Reference Determines The Method Called

```
public class Person {
    public void doPersonStuff() {
        ...
    }
}

public class Hero extends Person {
    public void doPersonStuff() {
        ...
    }
}

// Bob is a Person
bob.doPersonStuff();

// Clarke is a Hero
clark.doPersonStuff();
```

James Tam

New Terminology: Polymorphism

Poly = many Morphic = forms

- A polymorphic method has an implementation in the parent class and a different implementation in the child class.
- Polymorphism: the specific method called will be automatically determined without any type checking needed (the type of reference determines which method is called)
- Recall the example:

The Driver Class (Included For Reference)

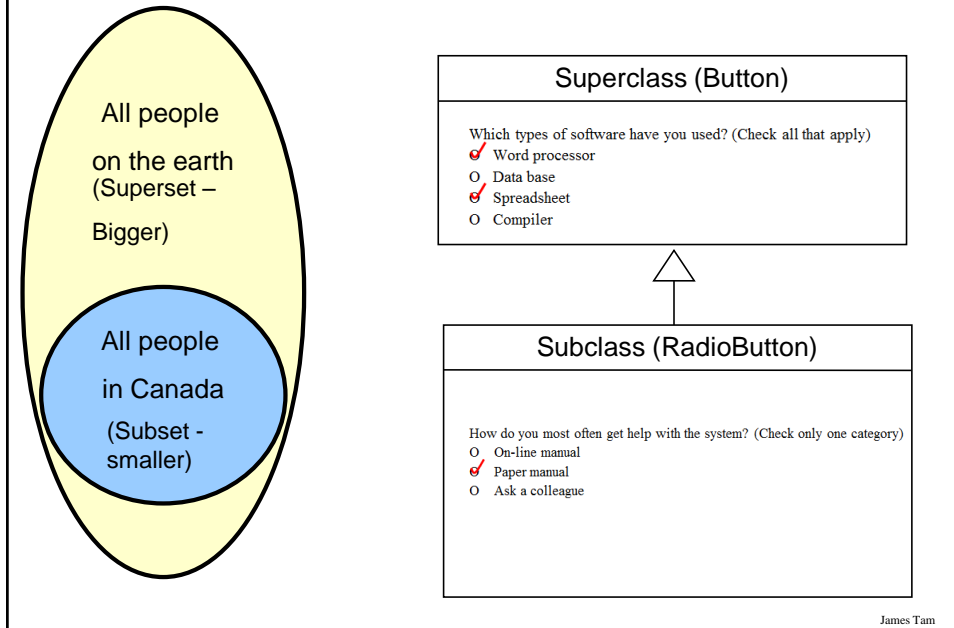
```
public class Driver
{
    public static void main(String [] args)
    {
        Person bob = new Person();
        bob.doPersonStuff();
        System.out.println();

        Hero clark = new Hero();
        clark.doPersonStuff();

        Pffff I need not go about mundane nicities such as eating!
    }
}
```

James Tam

New Terminology: Super-class Vs. Sub-class



The 'Super' Keyword

- Used to access the parts of the super-class.

- **Format:**

`<super>.<method or attribute>`

- **Example:**

```
public void doPersonStuff()
{
    System.out.println("Pffff I need not go about mundane"+
        " nicities such as eating!");

    super.doPersonStuff();
}
```

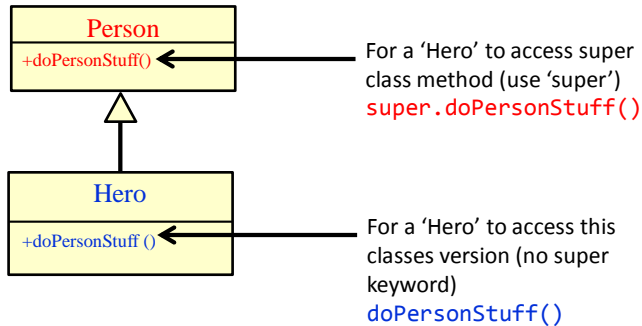
```
Pffff I need not go about mundane nicities such as eating!
Eat, sleep, drink, excrete, be fruitful
```

Parent's version of method

James Tam

Super Keyword: When It's Needed

- You only need this keyword when accessing non-unique methods or attributes (exist in both the super and sub-classes).

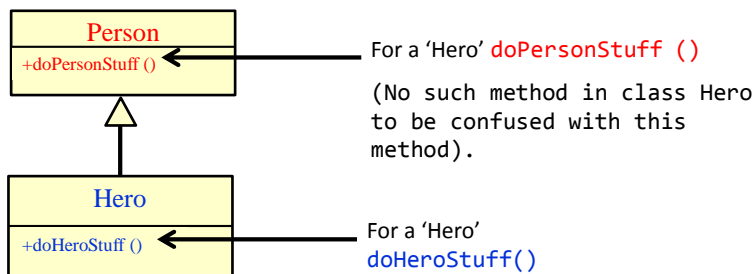


- Without the super keyword then the sub-class will be accessed

James Tam

Super Keyword: When It's Not Needed

- If that method or attribute exists in only one class definition then there is no ambiguity.



James Tam

Something Especially Good?

- Note: There Is No Super . Super In Java

James Tam

Using The Super Keyword

- Location of the complete example:
 - /home/219/examples/hierarchies/3super
 - Note: this example illustrated the use of the super keyword in conjunction with a method "doPersonStuff".
 - As long as access permissions allow it, *any attribute or method in the super class can be accessed in the same way* using the 'super' keyword.

James Tam

Class Hero: Using The Super Keyword

```
public class Hero extends Person
{
    public void doPersonStuff()
    {
        System.out.println("Pffff I need not go about mundane" +
            " nicities such as eating!");
        timeDelay();
        super.doPersonStuff();
        timeDelay();
        // Hero specific output
        System.out.println("...well actually I do :$");
    }
    private void timeDelay()
    {
        final long DELAY_TIME = 1999999999;
        for (long i = 0; i <= DELAY_TIME; i++);
    }
}
```

For any Person

```
public void doPersonStuff()
{
    System.out.println("Eat, sleep, drink, excrete, be" +
        " fruitful");
}
```

James Tam

Using The Super Keyword Again

- This fourth example illustrates the use of this keyword with the constructor and the (new to this example) toString() method
- Note how the toString() method delegates some behavior to the parent class and implements some of the behaviors in the child class.
- Location of the full example:
/home/219/examples/hierarchies/4superConstructors

James Tam

Class Person

```
public class Person {
    private int age;

    public Person() {
        age = 0;
    }

    public Person(int anAge) {
        age = anAge;
    }

    public void doPersonStuff() {
        System.out.println("Eat, sleep, drink, excrete, be" +
            " fruitful");
    }
}
```

James Tam

Class Person (2)

```
// NEW
public String toString()
{
    String s = "";
    s = s + "Age of the person: " + age;
    return(s);
}
}
```

James Tam

Class Hero: Using **Super()**

```
public class Hero extends Person
{
    private int heroicCount;

    public Hero() {
        super();
        heroicCount = 0;
    }

    public Hero(int anAge) {
        super(anAge);
        heroicCount = 0;
    }

    public void doHeroStuff() {
        ...
        heroicCount++;
    }
}
```

```
public Person() {
    age = 0;
}
```

```
public Person(int anAge) {
    age = anAge;
}
```

James Tam

Class Hero: Using **Super():2**

```
public String toString()
{
    String s = super.toString();
    if (s != null)
        s = s + "\n" + "Count of noble and heroic deeds " +
            heroicCount;
    return(s);
}
```

```
// Class Person
public String toString()
{
    String s = "";
    s = s + "Age of the person: " + age;
    return(s);
}
```

James Tam

The Driver Class

```
public class Driver
{
    public static void main(String [] args)
    {
        Person bob = new Person(55);
        Hero clark = new Hero(25);
    }
}
```

```
public Person(int anAge){
    age = anAge;
}
```

```
public Hero(int anAge){
    super(anAge);
    heroicCount = 0;
}
```

```
public Person(int anAge){
    age = anAge;
}
```

James Tam

The Driver Class: 2

```
System.out.println("Bob\n" + bob);
```

```
public String toString()
{
    String s = "";
    s = s + "Age of the person: " + age;
    return(s);
}
```

```
System.out.println("Clark\n" + clark);
```

```
}
}
```

```
public String toString() // Hero
{
    String s = super.toString();
    if (s != null)
        s = s + "\n" + "Count of noble and heroic deeds " +
            heroicCount;
    return(s);
}
```

```
public String toString() // Person
{
    String s = "";
    s = s + "Age of the person: " + age;
    return(s);
}
```

James Tam

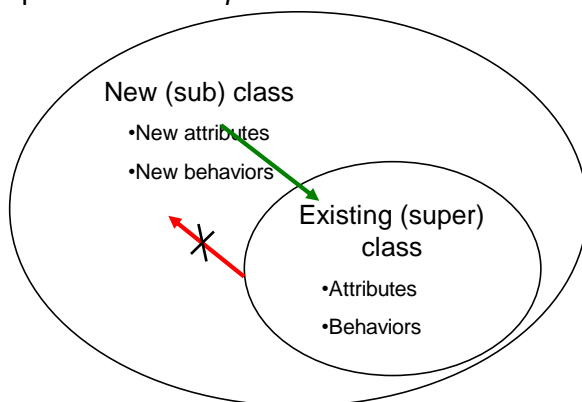
Example 4 Synopsis

- Using the super keyword to access the parent constructors
- User the super keyword to access the parent implementation of toString()
- Both method calls would delegate some of the required behaviors to the parent (access modify parent class attributes) and then the child implement the remaining behavior (access child class attributes)

James Tam

Keep In Mind: Inheritance Is A One Way Relationship!

- *A Hero is a Person but a Person is not a Hero!*
- That means that while the *sub-class can access the super-class* parts but the *super-class cannot access the sub-class* parts.



James Tam

Access Modifiers And Inheritance

- Private '-': still works as-is, private attributes and methods can only be accessed within that classes' methods.
 - Child classes, similar to other classes must access private attributes through public methods.
- Public '+': still works as-is, public attributes and methods can be accessed anywhere.
- **New level of access, Protected '#'**: can access the method or attribute in the class or its sub-classes.

James Tam

Summary: Levels Of Access Permissions

Access level	Accessible to		
	Same class	Subclass	Not a subclass
Public	Yes	Yes	Yes
Protected	Yes	Yes	No
Private	Yes	No	No

James Tam

Levels Of Access Permission: An Example

```
public class P
{
    private int num1;
    protected int num2;
    public int num3;
    // Can access num1, num2 & num3 here.
}

public class C extends P
{
    // Can't access num1 here
    // Can access num2, num3
}

public class Driver
{
    // Can't access num1 here and generally can't access num2
    // here
    // Can access num3 here
}
```

James Tam

General Rules Of Thumb

- Variable attributes should not have protected access but instead should be private.
- Most methods should be public.
- Methods that are used only by the parent and child classes should be made protected.

James Tam

Updated Scoping Rules

- When referring to an identifier in a method of a class
 1. Look in the local memory space for that method
 2. Look in the definition of the class
 3. New: Look in the definition of the parent class

James Tam

Updated Scoping Rules (2)

```
public class P
{
    <<< Third: Parent's attribute >>>
}
public class C extends P
{
    <<< Second: Attribute>>>

    public void method ()
    {
        <<< First: Local >>>
        Reference to an identifier e.g., 'num'
    }
}
```

Similar to how local variables can shadow attributes, the child attributes can shadow parent attributes.

James Tam

Updated Scoping Rules: A Trace

- Location of the full online example:
/home/219/examples/hierarchies/5scope

James Tam

Scoping Rules: Review Code (1 Class)

```
public class Driver {  
    public static void main(String [] args) {  
        System.out.println("REVIEW");  
        System.out.println("-----");  
        P p = new P();  
        p.method1();  
        System.out.println();  
    }  
}
```

```
public class P {  
    protected int x = 1;  
    private int y = 2;  
  
    public void method1() {  
        int x = 10;  
        int y = 20;  
        System.out.println("P.method1()");  
        System.out.println("Locals shadow attributes");  
        System.out.println("x/y: " + x + " " + y);  
    }  
}
```

```
P.method1()  
Locals shadow attributes  
x/y: 10 20
```

Scoping Rules: Review Code (1 Class): 2

```
p.method2();  
System.out.println();
```

```
public void method2()  
{  
    int x = 10;  
    int y = 20;  
    System.out.println("P.method2()");  
    System.out.println("Loc x/y: " + x + " " + y);  
    System.out.println("Attr x/y: " + this.x + " " +  
                        this.y);  
}
```

```
P.method2()  
Loc x/y: 10 20  
Attr x/y: 1 2
```

James Tam

Updated Scoping Rules

```
System.out.println("NEW: INHERITANCE HIERARCHIES");  
System.out.println("-----");  
C c = new C();  
c.method1();
```

```
// Child  
public class C extends P {  
    private int x = 3;  
    private int z = 4;  
  
    public void method1() {  
        System.out.println("C.method1()");  
        System.out.println("Child attributes");  
        System.out.println("x/z: " + this.x + " " +  
                            this.z);  
    }  
}
```

James Tam

Updated Scoping Rules (2)

```
c.method2();
```

```
// Child
public void method2() {
    int x = 100;
    int y = 200;
    int z = 300;
    System.out.println("C.method2()");
    System.out.println("Local shadows all");
    System.out.println("x/y/z: " + x + " " + y + " " +
        z);
}
```

James Tam

Updated Scoping Rules (3)

```
c.method3();
```

```
// Child
public void method3() {
    int x = 100;
    int y = 200;
    int z = 300;
    System.out.println("C.method3()");
    System.out.println("Loc x/y/z: " + x + " " + y + " " + z);

    System.out.println("P(x/y): " + super.x + " " + super.getY());
    // super.y : syntax error, access permission violated
    System.out.println("C(x/z): " + this.x + " " + this.z);
}
```

```
public class P
{
    protected int x = 1;
    private int y = 2;
}
```

James Tam

The Final Modifier (Inheritance)

- What you know: the keyword `final` means unchanging (used in conjunction with the declaration of constants)
- Methods preceded by the final modifier cannot be overridden
e.g., `public final void cannot_override()`
- Classes preceded by the final modifier cannot be extended
-e.g., `final public class CANT_BE_EXTENDED`

James Tam

Review: Casting

- The casting operator can be used to convert between types.
- **Format:**
`<Variable name> = (type to convert to) <Variable name>;`
- **Example (casting needed: going from more to less)**
`double full_amount = 1.9;
int dollars = (int) full_amount;`
- **Example (casting not needed: going from less to more)**
`int dollars = 2;
double full_amount = dollars;`

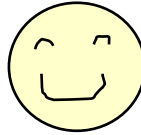
James Tam

Real Life Examples: Expectations Vs. Reality

Getting more than expected: acceptable

You are owed \$100

You receive \$1000



Getting less than expected: not acceptable

You are owed \$100

You receive \$10

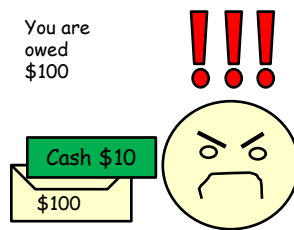


James Tam

Real Life Examples: Expectations Vs. Reality (2)

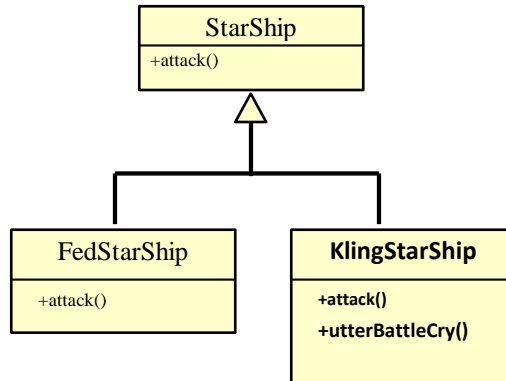
Misrepresenting reality: still not acceptable in the end

You are owed \$100



James Tam

Example Inheritance Hierarchy

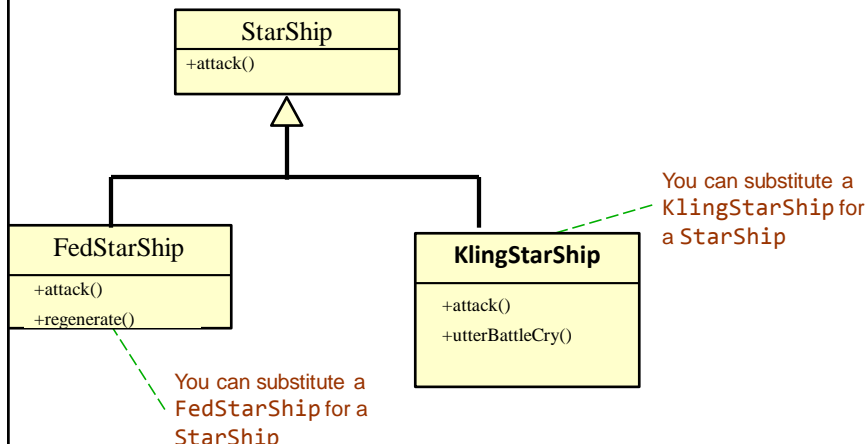


James Tam

Casting And Inheritance (Up)



- Because the child class IS-A parent class you can substitute instances of a subclass for instances of a superclass.

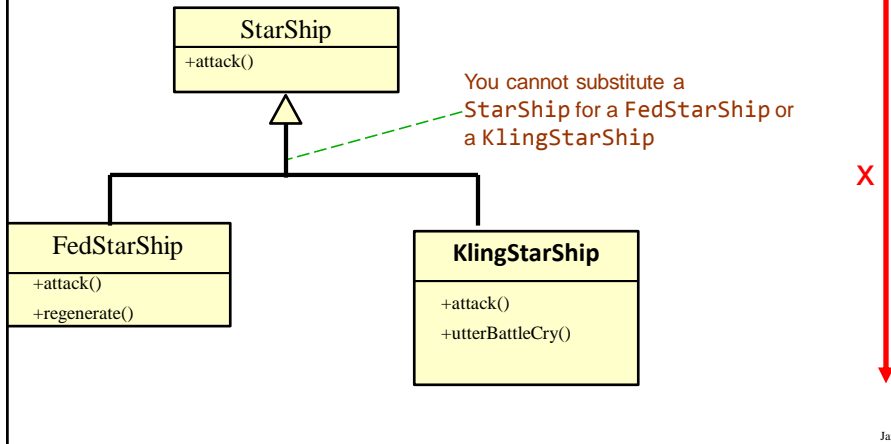


James Tam

Casting And Inheritance (Down)



- You cannot substitute instances of a superclass for instances of a subclass
 - Why?



Reminder: Operations Depends On Type

- Sometimes the same symbol performs different operations depending upon the type of the operands/inputs.
- Example:

```
int num1 = 2;
int num2 = 3;
num1 = num1 + num2;
```

Vs.

```
String aString = "foo" + "bar";
```
- Some operations won't work on some types
- Example:

```
String aString = 2 / 3;
```

James Tam

Reminder: Behavior Depends Upon Class Type

- The methods that can be invoked by an object depend on the class definition
- Example:

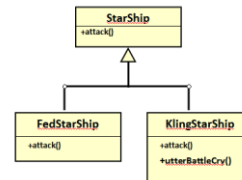
```
class X                class Y
{                      {
    method1() {        method2() {
    }                  }
}                      }
```

```
X x1 = new X();
x1.method1();        // Yes
Y y1= new Y();
y1.method1();        // No
```

James Tam

Casting And Inheritance

```
StarShip regular = new StarShip();
KlingStarShip kling = new KlingStarShip();
```



X `regular.utterBattleCry();` // Inappropriate action for type

```
regular = kling;
```

X `regular.utterBattleCry();` // I think I point to the wrong type
`((KlingStarShip) regular).utterBattleCry();` // I think I point
// to the correct
// type (NOT!)

```
regular = new StarShip();
```

X `kling = (KlingStarShip) regular;` // Dangerous cast
`kling.utterBattleCry();` // Inappropriate action for type

James Tam

Caution About Class Casting: Check First!

- When casting between classes only use the casting operator if you are sure of the type!
- Check if an object is of a particular type is via the instanceof operator
- (When used in an expression the instanceof operator returns a boolean result)
- Format:

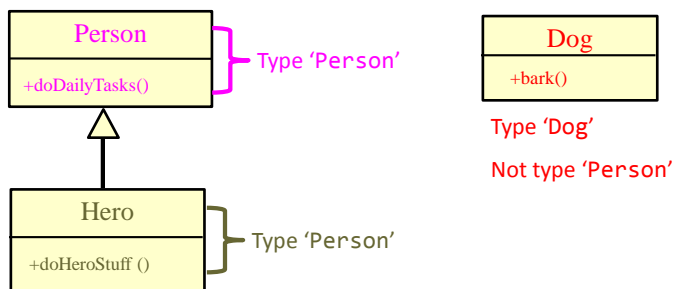
```
if (<reference name> instanceof <class name>)
```
- Example:

```
if (supPerson instanceof Person)
```

James Tam

Instanceof Example

- Location of the full example:
`/home/219/examples/hierarchies/6typeCheck`



James Tam

Driver.main()

```
Person regPerson = new Person();
Hero supPerson = new Hero();
Dog rover = new Dog();

// Instanceof checks if the object is a certain type or
// a subclass of that type (e.g., a Hero is a Person)
if (regPerson instanceof Person)
    System.out.println("regPerson is a type of Person");
if (supPerson instanceof Person)
    System.out.println("supPerson is also a type of Person");

// Checks for non-hierarchical: Compiler prevents nonsensical
// checks
//if (rover instanceof Person)
//    System.out.println("Rover is also a type of Person");
```

James Tam

Driver.main(): 2

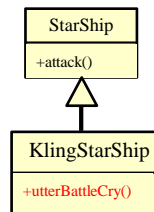
```
if (supPerson instanceof Hero)
    System.out.println("supPerson is a type of Hero");

// Checks within hierarchy: Compiler doesn't prevent
if (regPerson instanceof Hero)
    System.out.println("[Should never appear]: regPerson is a
        type of Hero");
```

James Tam

Containers: Homogeneous

- Recall that arrays must be homogeneous: all elements must be of the same type e.g., `int [] grades`
- **Again recall:** A child class is an instance of the parent (a more specific instance with more capabilities).



- If a container, such as an array is needed for use in conjunction with an inheritance hierarchy then the type of each element can simply be the parent.

```
StarShip [] array = new StarShip[2];
array[0] = new StarShip(); // [0] wants a StarShip, gets a StarShip
array[1] = new KlingStarShip(); // [1] wants a StarShip, gets a
// KlingStarShip (even better!)
```



James Tam

The Parent Of All Classes

- You've already employed inheritance.
- Class `Object` is at the top of the inheritance hierarchy.
- Inheritance from class `Object` is implicit.
- All other classes automatically inherit its attributes and methods (left and right are logically the same)

```
class Person          class Person extends Object
{
}
}                    }
```

-e.g., "`toString()`" are available to its child classes

- For more information about this class see the url:

<http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html>

James Tam

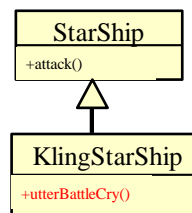
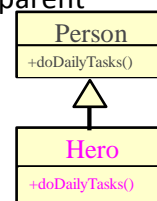
The Parent Of All Classes (2)

- This means that if you want to have an array that can contain *any type* of Java object then it can be an array of type `Object`.
 - `Object [] array = new Object[SIZE];`
- Built-in array-like classes such as class `Vector` (an array that 'automatically' resizes itself consists of an array attribute whose type is class `Object`)
 - For more information on class `Vector`:
 - <http://docs.oracle.com/javase/6/docs/api/java/util/Vector.html>

James Tam

Determining Type: Hierarchies

- As mentioned: normally type checking should not be needed for a polymorphic method (the child class overrides a parent method).
 - No `instanceof` needed
- However type checking is needed if a method specific to the child is being invoked.
 - Check with `instanceof` is needed



James Tam

Example: Containers With 'Different' Types

- Location of the full example:
- /home/219/examples/hierarchies/7hierarchiesContainment

James Tam

Driver Class: SpaceSimulator

```
public class SpaceSimulator
{
    public static void main(String [] args)
    {
        Galaxy alpha = new Galaxy();
        alpha.display();
        alpha.runSimulatedAttacks();
    }
}
```

James Tam

Class Galaxy

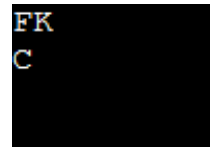
```
public class Galaxy {
    public static final int SIZE = 4;
    private StarShip [][] grid;
```

James Tam

Class Galaxy (2)

```
public Galaxy () {
    boolean squareOccupied = false;
    grid = new StarShip [SIZE][SIZE];
    int r;
    int c;
    int hull;

    for (r = 0; r < SIZE; r++) {
        for (c = 0; c < SIZE; c++)
        {
            grid[r][c] = null;
        }
    }
    grid[0][0] = new FedStarShip();
    grid[0][1] = new KlingStarShip();
    grid[1][0] = new StarShip();
}
```



James Tam

Class Galaxy (3)

```
public void runSimulatedAttacks() {
    int damage;
    damage = grid[0][0].attack();
    System.out.println("Fed ship attacks for: " + damage);
    System.out.println("<<< FedStarShip.attack() >>>");
    Fed ship attacks for: 66

    damage = grid[0][1].attack();
    System.out.println("Kling ship attacks for: " + damage);
    System.out.println("<<< KlingStarShip.attack() >>>");
    Kling ship attacks for: 116

    damage = grid[1][0].attack();
    System.out.println("Old style ship attacks for: " +
        damage);
    Old style ship attacks for: 50
    System.out.println();
}
```

Type check not needed because: `attack()` method is overridden / polymorphic

James Tam

FK
C

Class Galaxy (4)

```
/* Won't work because it's an array of references
   to StarShips not KlingStarShips.
   grid[1][0].utterBattleCry(); */

if (grid[0][0] instanceof KlingStarShip)
    ((KlingStarShip) grid[0][0]).utterBattleCry();
    Heghlu'meH QaQ jajvam!

if (grid[0][1] instanceof KlingStarShip)
    ((KlingStarShip) grid[0][1]).utterBattleCry();

if (grid[1][0] instanceof KlingStarShip)
    ((KlingStarShip) grid[1][0]).utterBattleCry();

}
} // End runSimulatedAttacks()
```

Type check 'instanceof' needed because: Array of StarShips but `utterBattleCry()` unique to `KlingStarShip`

James Tam

Class StarShip

```
public class StarShip {
    public static final int MAX_HULL = 400;
    public static final char DEFAULT_APPEARANCE = 'C';
    public static final int MAX_DAMAGE = 50;
    private char appearance;
    private int hullValue;

    public StarShip () {
        appearance = DEFAULT_APPEARANCE;
        hullValue = MAX_HULL;
    }

    public StarShip (int hull) {
        appearance = DEFAULT_APPEARANCE;
        hullValue = hull;
    }
}
```

James Tam

Class StarShip (2)

```
public StarShip (char newAppearance) {
    this();
    appearance = newAppearance;
}

public int attack() {
    System.out.println("<<< StarShip.attack() >>>");
    return(MAX_DAMAGE);
}
```

James Tam

Class StarShip (3): Get()'s, Set()'s

```
public char getAppearance () {
    return appearance;
}

public int getHullValue() {
    return(hullValue);
}

public void setAppearance(char newAppearance) {
    appearance = newAppearance;
}

public void setHull(int newHullValue) {
    hullValue = newHullValue;
}
}
```

James Tam

Class FedStarShip

```
public class FedStarShip extends StarShip {
    public static final int MAX_HULL = 800;
    public static final char DEFAULT_APPEARANCE = 'F';
    public static final int MAX_DIE_ROLL = 6;
    public static final int DIE_ROLL_BOOSTER = 1;
    public static final int NUM_DICE = 20;

    public FedStarShip() {
        super();
        setHull(MAX_HULL); // 800 not 400 due to shadowing
        setAppearance(DEFAULT_APPEARANCE); // 'F' not 'C'
    }
}
```

} Shadows
parent
constants

James Tam

Class FedStarShip (2)

```
// Overridden / polymorphic method
public int attack() {
    System.out.println("<<< FedStarShip.attack() >>>");
    Random aGenerator = new Random();
    int i = 0;
    int tempDamage = 0;
    int totalDamage = 0;

    for (i = 0; i < NUM_DICE; i++)
    {
        tempDamage = aGenerator.nextInt(MAX_DIE_ROLL) +
            DIE_ROLL_BOOSTER;
        totalDamage = totalDamage + tempDamage;
    }
    return(totalDamage);
}
}
```

James Tam

Class KlingStarShip

```
public class KlingStarShip extends StarShip {
    public static final char DEFAULT_APPEARANCE = 'K';
    public static final int MAX_DIE_ROLL = 12;
    public static final int DIE_ROLL_BOOSTER = 1;
    public static final int NUM_DICE = 20;

    public KlingStarShip() {
        super();
        setAppearance(DEFAULT_APPEARANCE);
    }
    // Unique to KlingStarShip objects
    public void utterBattleCry() {
        System.out.println("Heghlu'meH QaQ jajvam!");
    }
}
}
```

James Tam

Class KlingStarShip (2)

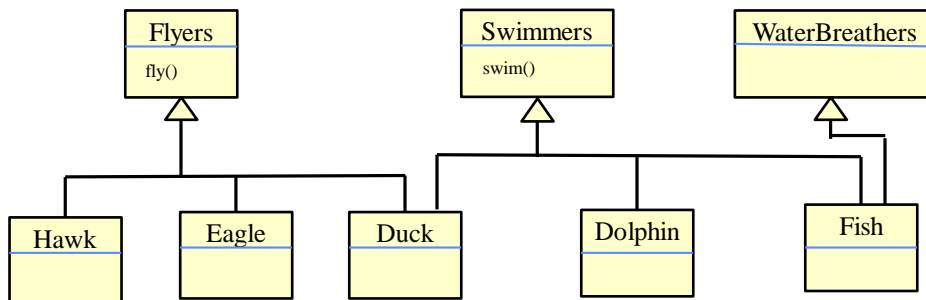
```
// Overridden / polymorphic method
public int attack() {
    System.out.println("<<< KlingStarShip.attack() >>>");
    Random aGenerator = new Random();
    int i = 0;
    int tempDamage = 0;
    int totalDamage = 0;

    for (i = 0; i < NUM_DICE; i++) {
        tempDamage = aGenerator.nextInt(MAX_DIE_ROLL) +
            1 * DIE_ROLL_BOOSTER;
        totalDamage = totalDamage + tempDamage;
    }
    return(totalDamage);
}
```

James Tam

Multiple Inheritance

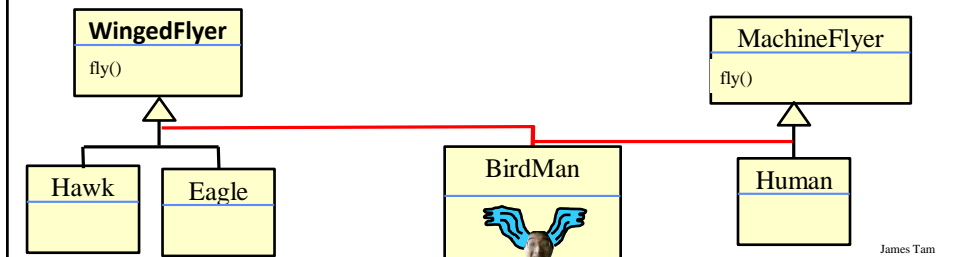
- What happens if some behaviors or attributes are common to a group of classes but some of those classes include behaviors shared with other groups?
- Or some groups of classes share some behaviors but not others?



James Tam

Multiple Inheritance (2)

- It is implemented in some languages e.g., C++
- It is not implemented in other languages e.g., Java
- Pro: It allows for more than one parent class
 - (JT: rarely needed but nice to have that capability for that odd exceptional case).
- Con: Languages that allow for multiple inheritance require a more complex implementation even for single inheritance (classes only have one parent) cases.



Java Interfaces

- Similar to a class
- Provides a design guide rather than implementation details
- Specifies what methods should be implemented but not how
 - An important design tool: Agreement for the interfaces should occur very early before program code has been written.
 - (Specify the signature of methods so each part of the project can proceed with minimal coupling between classes).
 - Changing the method body rather than the method signature won't 'break' code.
- It's a design tool so interfaces cannot be instantiated
 - Q: What if one could instantiate an interface directly?

James Tam

Interfaces: Format

Format for defining an interface

```
public interface <name of interface>
{
    constants
    methods to be implemented by the class that realizes this
        interface
}
```

Format for realizing / implementing the interface

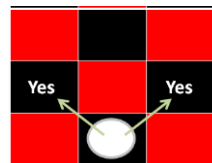
```
public class <name of class> implements <name of interface>
{
    attributes
    methods actually implemented by this class
}
```

James Tam

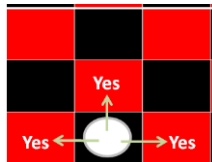
Interfaces: A Checkers Example



Regular board¹



Regular rules²



Variant rules²

¹ From www.allaboutfungames.com

² Board images from

James Tam

Interface Board

```
public interface Board
{
    public static final int SIZE = 8;
    public void displayBoard();
    public void initializeBoard();
    public void movePiece();
    boolean moveValid(int xSource,
                      int ySource,
                      int xDestination,
                      int yDestination);
    ...
}
```

James Tam

Class RegularBoard

```
public class RegularBoard implements Board
{
    public void displayBoard()
    {
        ...
    }

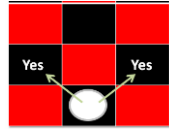
    public void initializeBoard()
    {
        ...
    }
}
```

James Tam

Class RegularBoard (2)

```
public void movePiece() {
    // Get (x, y) coordinates for the source and destination
    if (moveValid(xS, yS, xD, yD) == true)
        // Actually move the piece
    else
        // Don't move piece and display error message
}

public boolean moveValid(int xSource, int ySource,
                          int xDestination,
                          int yDestination)
{
    if (moving forward diagonally)
        return(true);
    else
        return(false);
}
} // End of class RegularBoard
```



James Tam

Class VariantBoard

```
public class VariantBoard implements Board
{
    public void displayBoard ()
    {
        ...
    }

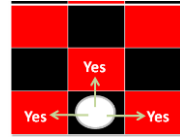
    public void initializeBoard ()
    {
        ...
    }
}
```

James Tam

Class VariantBoard (2)

```
public void movePiece() {
    // Get (x, y) coordinates for the source and destination
    if (moveValid (xS, yS, xD, yD) == true)
        // Actually move the piece
    else
        // Don't move piece and display error message
    }

    public boolean moveValid(int xSource, int ySource,
                               int xDestination,
                               int yDestination)
    {
        if (moving straight-forward or straight side-ways)
            return(true);
        else
            return(false);
    }
} // End of class VariantBoard
```



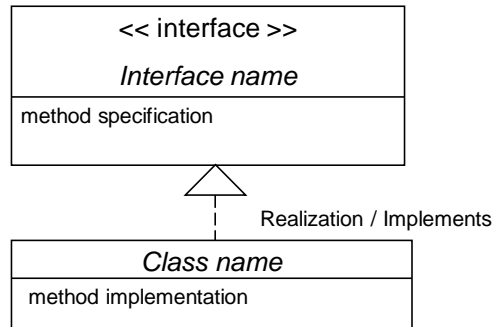
James Tam

Interfaces: Recapping The Example

- Interface Board
 - No state (variable data) or behavior (body of the method is empty)
 - Specifies the behaviors that a board *should* exhibit e.g., clear screen
 - This is done by listing the methods that must be implemented by classes that implement the interface.
- Class RegularBoard and VariantBoard
 - Can have state and methods
 - They must implement all the methods specified by the interface 'Board' (but can also implement other methods too)

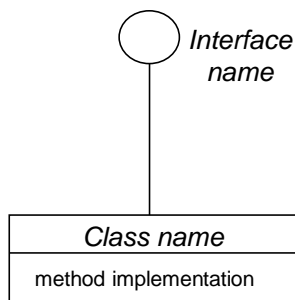
James Tam

Specifying Interfaces In UML



James Tam

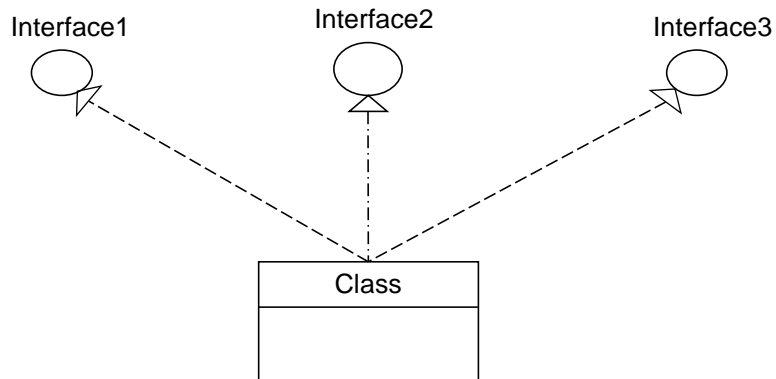
Alternate UML Representation (Lollipop Notation)



James Tam

Implementing Multiple Interfaces

- Java allows for this.



James Tam

Implementing Multiple Interfaces

Format:

```
public class <class name> implements <interface name 1>,  
    <interface name 2>, <interface name 3>...  
{  
  
}
```

James Tam

Multiple Implementations Vs. Multiple Inheritance

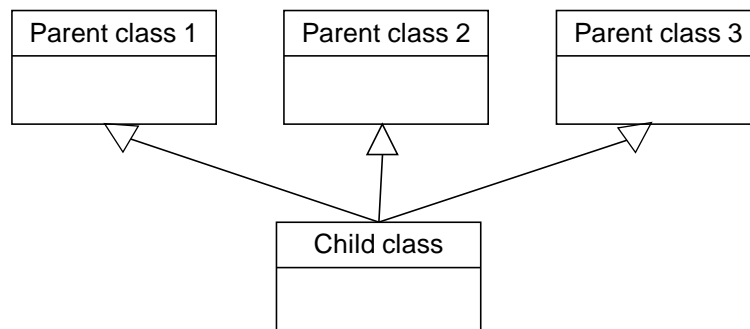
- A class can implement multiple interfaces
- Classes in Java cannot extend more than one class
- Again: multiple inheritance is **not possible in Java** but is possible in other languages such as C++:
 - Multiple inheritance (mock up code)

```
class <class name 1> extends <class  
name 2>, <class name 3>...  
{  
  
}
```

James Tam

Multiple Implementations Vs. Multiple Inheritance (2)

- Multiple inheritance: conceptual view representing using UML



James Tam

Abstract Classes (Java)

- Classes that cannot be instantiated.
- A hybrid between regular classes and interfaces. Some methods may be implemented while others are only specified (no body).
- Used when the parent class:
 - specifies a default implementation of some methods,
 - but cannot define a complete default implementation of other methods (implementation must be specified by the child class).

- **Format:**

```
public abstract class <class name>
{
    <public/private/protected> abstract method ();
}
```

James Tam

Abstract Classes (Java): 2

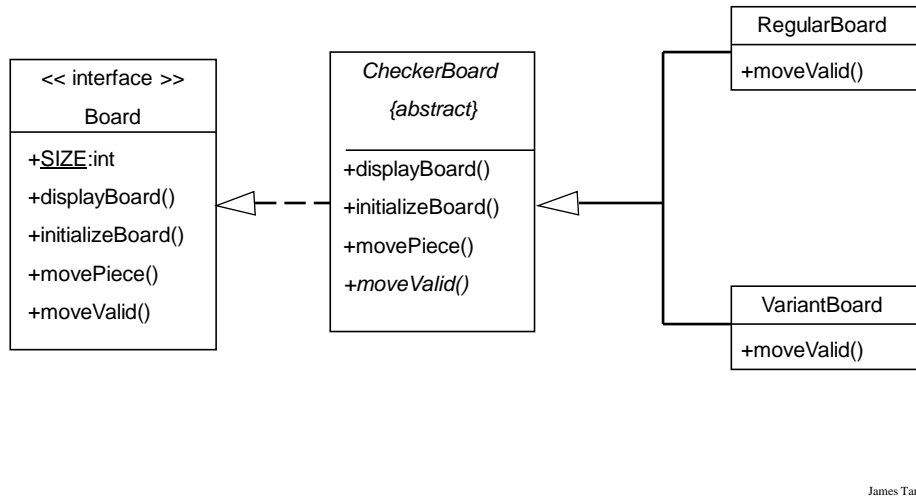
- **Example¹:**

```
public abstract class BankAccount
{
    protected float balance;
    public void displayBalance()
    {
        System.out.println("Balance $" + balance);
    }
    public abstract void deductFees() ;
}
```

1) From "Big Java" by C. Horstmann pp. 449 – 500.

James Tam

Another Example For Using An Abstract Class



You Should Now Know

- What is inheritance, when to employ it, how to employ it in Java
- How casting works within an inheritance hierarchy
 - When the `instanceof` operator should and should not be used to check for type in an inheritance hierarchy
- What is the effect of the keyword "final" on inheritance relationships
- What is method overriding
 - How does it differ from method overloading
 - What is polymorphism
- What are the benefits of employing inheritance

James Tam

You Should Now Know (2)

- How does the 'protected' level of access permission work, how do `private` and `public` access permissions work with an inheritance hierarchy.
 - Under what situations should each level of permission be employed
- Updated scoping rules (includes inheritance) and how shadowing works with an inheritance hierarchy
- How the 'super' keyword works, when it is and is not needed
- Class `Object` is the parent of all classes in Java
 - Capabilities inherited from the parent (if you refer to the API for class `Object`)
- How homogeneous composite types (such as arrays) can appear to contain multiple types within one container

James Tam

You Should Now Know (3)

- What are interfaces/types
 - How do types differ from classes
 - How to implement and use interfaces in Java
 - When interfaces should be employed
- What are abstract classes in Java and how do they differ from non-abstract classes and interfaces.
 - When to employ abstract classes vs. interfaces vs. 'regular' classes
- How to read/write UML notations for inheritance and interfaces.
- What is the difference between early and late binding
- What is multiple inheritance
 - How does it differ from multiple implementations
 - What are its advantages and disadvantages

James Tam

Copyright Notification

- “Unless otherwise indicated, all images in this presentation are used with permission from Microsoft.”